DTIC FILE COPY

(4)

SEC

REPORT DOCUMENTATION PAGE

1a. AD-A200 894	1b RESTRICTIVE MARKINGS None
2a.	3. DISTRIBUTION/AVAILABILITY OF REPORT Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) 144-X125/144-X260	5. MONITORING ORGANIZATION REPORT NUMBER(S) 4331 636 (R&T Project Code)
6a NAME OF PERFORMING ORGANIZATION University of Wisconsin; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Computer Science Division Attn: Dr. Ralph Wachter, Code 1133
6c. ADDRESS (City, State, and ZIP Code) Research Administration - Financial 750 University Avenue Madison, Wisconsin 53706	7b. ADDRESS (City, State, and ZIP Code) Office of Naval Research 800 North Quincy Street Arlington, VA 22217-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA; 8b. OFFICE SYMBOL (If applicable) DARPA	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-85-K0788
8c. ADDRESS (City, State, and ZIP Code) 1400 North Wilson Blvd. Arlington, VA 22209	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
61153N 14	RR01408	RR0140801	NR049-636

11. TITLE (Include Security Classification) Extensible Database Systems/Charlotte Project

12. PERSONAL AUTHOR(S) DeWitt, David Johns -- Solomon, Marvin H.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final Technical	FROM 8-21-85 TO 01-15-88		

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Man-Machine Relations
0508			Computer and Realted Programming
0902			

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

See Attached.

DTIC ELECTE OCT 2 6 1988 S D

D

DISTRIBUTION [illegible] Approved [illegible] Distr[illegible]

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Dr. Ralph Wachter (D. DeWitt, M. Solomon)	22b TELEPHONE (Include Area Code) 202-696-4303; 22c. OFFICE SYMBOL 1133

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete — SECURITY CLASSIFICATION OF THIS PAGE

*U.S. Government Printing Office: 1985-507-047

FINAL REPORT FOR EXTENSIBLE DATABASE SYSTEMS

Michael J. Carey
David J. DeWitt

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

1. INTRODUCTION

Until recently, research and development efforts in the database management systems area were focused on supporting traditional business applications. The design of database systems capable of supporting non-traditional application areas, including engineering applications for CAD/CAM and VLSI data, scientific and statistical applications, expert database systems (which can really be viewed as adding database system technology to an AI application), and image/voice applications, has now emerged as an important new direction for database system research. These new applications differ from more conventional applications like transaction processing and record keeping in a number of important areas:

(1) **Data modeling requirements.** Each new application area requires a different set of data modeling tools. The types of entities and relationships that must be described for a VLSI circuit design are quite different from the data modeling requirements of a banking application.

(2) **Processing functionality.** Each new application area has a specialized set of operations that must be supported by the database system. Consider, for example, a database containing satellite images. It makes little sense to talk about doing joins between these images or even components of a single image. Instead, we are more likely to be interested in analyzing the image using specialized image processing and/or recognition algorithms. For example, if we are looking for crop diseases we will want to apply an algorithm that examines the images for the crop disease signatures. As another example, if we wanted to look for particular types of ships, we would be interested in using an image recognition algorithm to compare the satellite images with those of the relevant types of ships. As we will discuss in more detail below, we contend that it does not make sense to implement such algorithms in terms of relational database operations (or CODASYL or network data model primitives either).

(3) **Concurrency control and recovery mechanisms.** Each new application area also has slightly different requirements for concurrency control and recovery mechanisms. While locking and logging are the accepted mechanisms for conventional database applications, a versioning mechanism looks most appropriate for engineering applications. For image databases, which tend to be principally accessed in a read-only fashion, perhaps no concurrency control or recovery mechanism is needed.

(4) **Access methods and storage structures.** Each new application area also has dramatically different requirements for access methods and storage structures. Access and manipulation of VLSI databases is facilitated by new access methods such as R-Trees. Storage of image data is greatly simplified if the database system supports large multidimensional arrays as a basic data type (a capability provided by no commercial database system at this time). Storing such images as tuples in a relational database system is generally either impossible or terribly inefficient.

Keywords: EXODUS, [illegible]

88 1025 018

The EXODUS project at the University of Wisconsin [Care85, Care86a, Care86b, Grae87, Rich87]] is addressing the problems posed in these emerging applications by providing tools that will enable the rapid implementation of high-performance, application-specific database systems. EXODUS provides a set of kernel facilities for use across all applications, such as a versatile storage manager and a general-purpose manager for type-related dependency information. In addition, EXODUS provides a set of tools to help the database implementor (DBI) to develop new database system software. The implementation of some DBMS components is supported by tools which actually generate the components from specifications; for example, tools are provided to generate a query optimizer from a rule-based description of a data model, its operators, and their implementations. Other components, such as new abstract data types, access methods, and database operations, must be explicitly coded by the DBI due to their more widely-varying and highly algorithmic nature.[1] EXODUS attempts to simplify this aspect of the DBI's job by providing a set of high-leverage programming language constructs for the DBI to use in writing the code for these components.

In Section 2, we describe the architecture of EXODUS in more detail. The current status of the project is presented in Section 3 along with a list of the technical accomplishments of the project. A listing of the technical reports produced as part of the project is contained in Section 4.

2. AN OVERVIEW THE EXODUS ARCHITECTURE

Since EXODUS is basically a collection of components and tools that can be used in a number of different ways, describing EXODUS is more difficult than describing the structure/organization of other extensible database system designs that have appeared recently. We believe that the flexibility provided by the EXODUS approach will make the system usable for a much wider variety of applications (as we will discuss later).

The fixed components of EXODUS include the EXODUS *Storage Object Manager*, for managing persistent objects, and a generalized *Dependency Manager* (formerly called the *Type Manager*), for keeping track of information about various type-related dependencies. In addition to these fixed components, EXODUS also provides tools to aid the DBI in the construction of application-specific database systems. One such tool is the *E programming language*, which is provided for developing new database software components. A related resource is a *type-*

[1]Actually, EXODUS will provide a library of generally useful components, such as widely-applicable access methods including B+ trees and some form of dynamic hashing, but the DBI must implement components that are not available in the library.

independent module library; E's generator classes and iterators can be used to produce useful modules (e.g., various access methods) that are independent of the types of the objects on which they operate, and these modules can then be saved away for future use. Another class of tools are provided for *generating* components from a specification. An example is the EXODUS rule-based *Query Optimizer Generator*. We also envision providing similar generator tools to aid in the construction of the front-end portions of an application-specific DBMS. The components of EXODUS are described further in the following sections. More detail on the Storage Object Manager can also be found in [Care86a], the E programming language is described in [Rich87], and details regarding the Query Optimizer Generator and an initial evaluation of its performance can be found in [Grae87].

2.1. The Storage Object Manager

The Storage Object Manager provides *storage objects* for storing data and *files* for logically and physically grouping storage objects together. Also provided are a powerful buffer manager that buffers variable-length pieces of large storage objects, primitives for managing versions of storage objects, and concurrency control and recovery services for operations on storage objects and files.

A storage object is an uninterpreted container of bytes which can be as small (e.g., a few bytes) or as large (e.g., hundreds of megabytes) as demanded by an application. The distinction between small and large storage objects is hidden from higher layers of EXODUS software. Small storage objects reside within a single disk page, whereas large storage objects occupy potentially many disk pages. In either case, the object identifier (OID) of a storage object is an address of the form `(page #, slot #)`. The OID of a small storage object points to the object on disk; for a large storage object, the OID points to its *large object header*. A large object header can reside on a slotted page with other large object headers and small storage objects, and it contains pointers to other pages involved in the representation of the large object. Pages in a large storage object are private to that object (although pages are shared between versions of a large storage object). When a small storage object grows to the point where it can no longer be accommodated on a single page, the Storage Object Manager will automatically convert it into a large storage object, leaving its header in place of the original small object.

All read requests specify an OID and a range of bytes; the desired range of bytes is read into a contiguous region in the buffer pool (even if the bytes are distributed over several partially full pages on disk), and a pointer to the bytes is returned to the caller. Bytes may be overwritten directly, using this pointer, and a call is provided to tell the Storage Object Manager that a subrange of the bytes that were read have been modified (information needed for recovery to take place). For shrinking/growing storage objects, calls to insert bytes into and delete bytes from a specified offset within a storage object are provided, as is a call to append bytes to the end of an object. To make these operations efficient, large storage objects are represented using a B+ tree structure to index data pages on byte offset.

The Storage Object Manager also provides support for versions of storage objects. In the case of small storage objects, versioning is implemented by making a copy of the entire object before applying the update. Versions of large storage objects are maintained by copying and updating only those pages that differ from version to version. The Storage Object Manager also supports the deletion of a version with respect to a set of other versions with which it may share pages. The reason for only providing a primitive level of version support is that different EXODUS applications may have widely different notions of how versions should be supported. We do not omit version management altogether for efficiency reasons — it would be prohibitively expensive, both in terms of storage space and I/O cost, if clients were required to maintain versions of large objects externally by making entire copies.

For concurrency control, two-phase locking of byte ranges within storage objects is used, with a "lock entire object" option being provided for cases where object-level locking will suffice. To ensure the integrity of the internal pages of large storage objects during insert, append, and delete operations (e.g., while their counts and pointers are being changed), non-two-phase B+ tree locking protocols are employed. For recovery, small storage objects are handled by logging changed bytes and performing updates in place at the object level. Recovery for large storage objects is handled using a combination of shadowing and logging — updated internal pages and leaf blocks are shadowed up to the root level, with updates being installed atomically by overwriting the old object header with the new one. A similar scheme is used for versioned objects, but the before-image of the updated large object header (or entire small object) is retained as an old version of the object.

Finally, the Storage Object Manager provides the notion of a *file object*. A file object is an unordered set of related storage objects, and is useful in several different ways. First, the Storage Object Manager provides a mechanism for sequencing through all of the objects in a file, so related objects can be placed in a common file for sequential scanning purposes. Second, objects within a given file are placed on disk pages allocated to the file, so file objects provide support for objects that need to be co-located on disk.

2.2. The Dependency Manager

The EXODUS Dependency Manager is a repository for information related to persistent types.[2] It maintains information about all of the pieces (called *fragments*) that make up a compiled query, including type definitions and other E code, and about their relationships to one another. It also keeps track of the relationship between files and their types (by treating files in a manner similar to fragments). In short, the Dependency Manager keeps track of dependencies between types and most everything else that is related to or dependent upon such information.

More specifically, certain time ordering constraints must hold between the fragments constituting a complete query. For example, a compiled query plan must have been created more recently than the program text for any of the types or database operations that it employs, as otherwise out-of-date code will have been used in its creation. A given abstract data type, or a set of operations, is also likely to have multiple representations (e.g., E source code, an intermediate representation, and a linkable object file), and similar time ordering constraints must hold between these representations. The Dependency Manager's role is thus similar to the Unix™ `make` facility [Feld79]. Unlike `make`, which only examines dependencies and timestamps when it is started up, the Dependency Manager maintains a graph of inter-fragment dependencies at all times (and updates it incrementally).

The Dependency Manager also plays a role in maintaining data abstraction that distinguishes it from `make`. In particular, a given type used by a query plan is likely to use other types to constitute its internal representation. Strictly speaking, the first tvpe is not *dependent* upon the linkable object code of its constituent types' operations; that is, while it must eventually be linked with their code, it is not necessary that their object code be up to date, or even compiled, until link time. We call fragments of this sort *companions*; `make` has no facilities for specifying and using companions. The Dependency Manager requires such a facility, as otherwise it would be unable to provide a complete list of the objects constituting the compiled access plan for a query, which is necessary when a query is to be linked.

The Dependency Manager maintains the correct time ordering of fragments via two mechanisms, *rules* and *actions*. The set of fragments constitutes the nodes of an acyclic directed graph; rules generate the arcs of this graph. When a fragment is found to be older than those fragments upon which it depends (with the dependencies being determined from the rules), a search is made for an appropriate action that can be performed to bring the

[2] As we will explain shortly, new types in EXODUS are defined using the `class` and `dbclass` constructs of the E programming language.

fragment up to date. Both rules and actions are defined using a syntax based on regular expressions to allow a wide range of default dependencies to be specified conveniently.

2.3. The E Programming Language

A major tool provided by EXODUS is the E programming language and its compiler. E is a extension of C++ [Stro86] that aids the DBI in a number of problem areas related to database system programming, including interaction with persistent storage, accommodation of missing type (class) information, and query compilation. E is designed to be upward compatible with C++, and its extensions include both new language features and a number of predefined classes.

E was designed with the following database system architecture in mind: First, all access methods, data model operators, and utility functions are written in E. In addition to these modules, the Storage Object Manager, and the Dependency Manager, the database system includes the E compiler itself. At run time, database schema definitions (e.g., create relation commands) and queries are first translated into E programs and then compiled. One result of this architecture is a system in which the "impedance mismatch" [Cope84] between type systems disappears. Another is that the system is easy to extend. For example, the DBI may add a new data type by implementing it as an E class, storing its definition and implementation in files, and registering the resulting module with the Dependency Manager for later use.

The following paragraphs describe some of the more important features of E from the standpoint of the DBI. More details can be found in [Rich87].

2.3.1. Generator Classes for Unknown Types

One of the problems faced by the DBI is that many of the types involved in database processing are not known until well after the code needing those types is written. For example, the code implementing a hash-join algorithm does not know what types of entities it will have to join. Similarly, index code does not know what types of keys it will contain nor what type of entities it will index.

To address this problem, E augments C++ with *generator* classes, which are very similar to the parameterized clusters of CLU [Lisk77]. Such a class is parameterized in terms of one or more unknown types; within the class definition, these (formal) type names are used freely as regular type names. This mechanism allows one to define, for example, a class of the form `stack[ T ]` where the specific type (class) `T` of the stack elements is not known. The user of such a class must *instantiate* it by providing specific parameters to the class; e.g., one may declare `x` to be an integer stack via the declaration `stack[ int ] x`. Similarly, the DBI can define the type of a B+ tree node as a class in which both the key type and the type of entity being indexed are class parameters. Later, when the user builds an index over employees on social security number, the system generates and compiles a small E fragment which instantiates `BTnode[ SSN_type, EMP_type ]`. Such instantiation can be efficiently accomplished via a linking process [Atki78].

2.3.2. Class fileof[T] for Persistent Storage

Another problem in database system programming is that most file systems provide the DBI only with untyped storage. Thus, after being read from disk, all data must be explicitly type cast in the DBI's code before it can be operated upon. In addition, since the data resides on secondary storage, the DBI must include explicit calls to the buffer manager in order to use it. These factors increase the amount of code that the DBI must write, and they also provide increased opportunities for coding errors.

E's answer to this problem is the "built-in" generator class `fileof[ T ]` where `T` must be a `dbclass`. A dbclass is declared in the same way as a C++ class with the restriction that a dbclass may contain only other dbclasses. (Predefined dbclasses exist for the fundamental types int, float, char, etc.) Dbclasses were introduced so that the compiler can always distinguish between objects residing only on the heap and those that generally reside on disk (but may also reside in memory) since the implementation of the two is very different.

Instances of the `fileof` generator class are implemented as a descriptor (in memory) associated with a physical file (on disk). This implementation is hidden behind an operational interface that allows the user to bind typed pointers to objects in a file, to create and destroy objects in a file, etc. For example, the following function returns the sum of all the integers in a file of integers. (The file is passed by reference.)

```
int filesum( fileof[dbint]& f )
{
    dbint *p;   /* dbint is the predefined dbclass for int */
    int sum = 0;
    for( p = f.getfirst(); p != 0; p = f.getnext( p ) ) sum += *p;
    return sum;
}
```

Although this example is extremely simple, it illustrates the two features mentioned above. The first is that no casting is needed to use the integer pointer `p`; the second is that no buffer calls are necessary to access the objects in file `f`. Clearly, an important research direction related to the implementation of E is the optimization of the calls to the buffer manager generated by the E compiler (especially for files containing very large objects such as images).

2.3.3. Iterators for Scans and Query Processing

A typical approach for structuring a database system is to include a layer which provides *scans* over objects in the database. A scan is a control abstraction which provides a state-saving interface to the "memoryless" storage systems calls; this interface is needed for the record-at-a-time processing done in higher layers. A typical implementation of scans will allocate a data structure, called a *scan descriptor*, to save all needed state between calls; it is up to the user to pass the descriptor with every call.

The control abstraction of a scan is provided in EXODUS via the notion of an *iterator* [Lisk77, OBri86]. An iterator is a coroutine-like function that saves its data and control states between calls; each time the iterator produces (`yields`) a new value, it is suspended until resumed by the client. Thus, no matter how complicated the iterator may be, the client only sees a steady stream of values being produced. Finally, for implementation reasons, the client can only invoke an iterator within a new kind of structured statement, the `iterate` loop (which generalizes the `for ... in` loop of CLU).

The general idea for implementing scans should now be clear. For example, to implement a scan over B+ trees, we would write an iterator function which takes a B+ tree, a lower bound, and an upper bound as arguments. It would begin by searching down to the leaf level of the tree for the lower bound, keeping a stack of node pointers along the way. It would then walk the tree, yielding object references one at a time, until the upper bound is reached. At that point, the iterator would terminate.

Iterators are also used to piece executable queries together from a parse tree. If we consider a query to be a pipeline of processing filters, then each stage can be implemented as an iterator which is a client of one or more iterators (upstream in the pipe) and which yields its results to the next stage (downstream in the pipe). Execution of the pipeline will be demand-driven in nature. For example, the DBI for a relational DBMS would write code for select, project, and join as iterators implementing filters. Given the parse tree of a user query, it is a fairly simple task to produce E code that implements the pipeline.

2.4. Type-Independent Access Methods and Operator Methods

Layered above the Storage Object Manager is a collection of access methods that provide associative access to files of storage objects and further support for versioning (if desired). For access methods, EXODUS will provide a library of type-independent index structures including B+ trees, Grid files [Niev84], and linear hashing [Litw80]. These access methods will be implemented using the class generator and iterator capabilities provided by the E programming language. This capability enables existing access methods to be used with DBI-defined abstract data types without modification — as long as the capabilities provided by the data type satisfy the requirements of the access methods. In addition, a DBI may wish to implement new types of access methods in the process of developing an application-specific database system. EXODUS provides mechanisms to greatly simplify this task. First, since new access methods are written in E, the DBI is shielded from having to map main memory data structures onto storage objects and from having to write code to deal with buffering. E will also simplify the task of handling concurrency control and recovery for new access methods.

Layered above the access methods is a set of operator methods that implement the operations of the application's chosen data model. As for access methods, the class generator and iterator facilities of E facilitate the development of operator methods. Generally useful methods (e.g., selection) will be made available in a type-independent library; methods specific to a given application domain will have to be developed by the DBI.

2.5. The Rule-Based Query Optimizer Generator

Since we expect that EXODUS will be used for a wide variety of applications, each with a potentially different query language, it is not possible for EXODUS to furnish a single generic query language, and it is accordingly impossible for a single query optimizer to suffice for all applications. As an alternative, a generator for producing query optimizers for algebraic query languages has been implemented. The input to the query optimizer generator is a collection of rules regarding the operators of the target query language, the transformations that can be legally applied to these operators (e.g., pushing selections before joins), and a description of the methods that can be used to execute each operator in the query language (including their costs and side effects). The Query Optimizer Generator transforms these description files into C source code[3], producing an optimizer for the application's query language. Later, to optimize queries using the resulting optimizer, a query is first parsed and converted into

[3]Note: While E is the language that the DBI will use to implement a DBMS, we are implementing the various components of EXODUS in C.

its initial form as a tree of operators; it is then transformed by the generated optimizer into an optimized execution plan expressed as a tree of methods. During the process of optimizing a query, the optimizer avoids exhaustive search by using AI search techniques and employing past (learned) experience to direct the search. As described above, each method in the tree produced by the optimizer is implemented as an iterator generator in E. Thus, a post-optimization pass over the plan tree is made to produce E code corresponding to the plan. For queries involving more than one operator, the iterators are nested in a manner that allows the query to be processed in a pipelined fashion, as mentioned earlier.

2.6. Application-Specific DBMS Development

Figure 1 presents a sketch of the architecture of a functionally complete, application-specific database system implemented using EXODUS. The components in Figure 1 that are implemented by the DBI in E are the access methods and operator methods. As discussed above, EXODUS provides a library of type-independent access methods, so it might not be necessary for a DBI to actually implement any access methods. EXODUS will also provide a library of methods for a number of operators that operate on a single type of storage object (e.g., selection), but it will not provide application or data model specific methods. For example, since a method for examining objects containing satellite image data for the signature of a particular crop disease would not be useful in general, it does not belong in such a library. In general, the DBI will need to implement (using E) one or more methods for each operator in the query language associated with the target application. The DBI must also write code (i.e., dbclass member functions) for the operations associated with each new abstract data type that he or she wishes to define.

To clarify by using a familiar example, a DBI who wanted to implement a relational DBMS for business applications via the EXODUS approach would have to obtain code for the desired access methods (e.g., B+ trees and linear hashing) by extracting existing code from the library and/or by writing the desired code from scratch in E. Similarly, code must be obtained for the operator methods (e.g., relation scan, indexed selection, nested loops join, merge join, etc.) and for various useful types (e.g., date and money). A DBI implementing a database management system for an image application would have to implement an analogous set of routines, presumably including various spatial index structures, operations that manipulate collections of images, and an appropriate set of types. As discussed earlier, E is provided to greatly simplify these programming tasks.

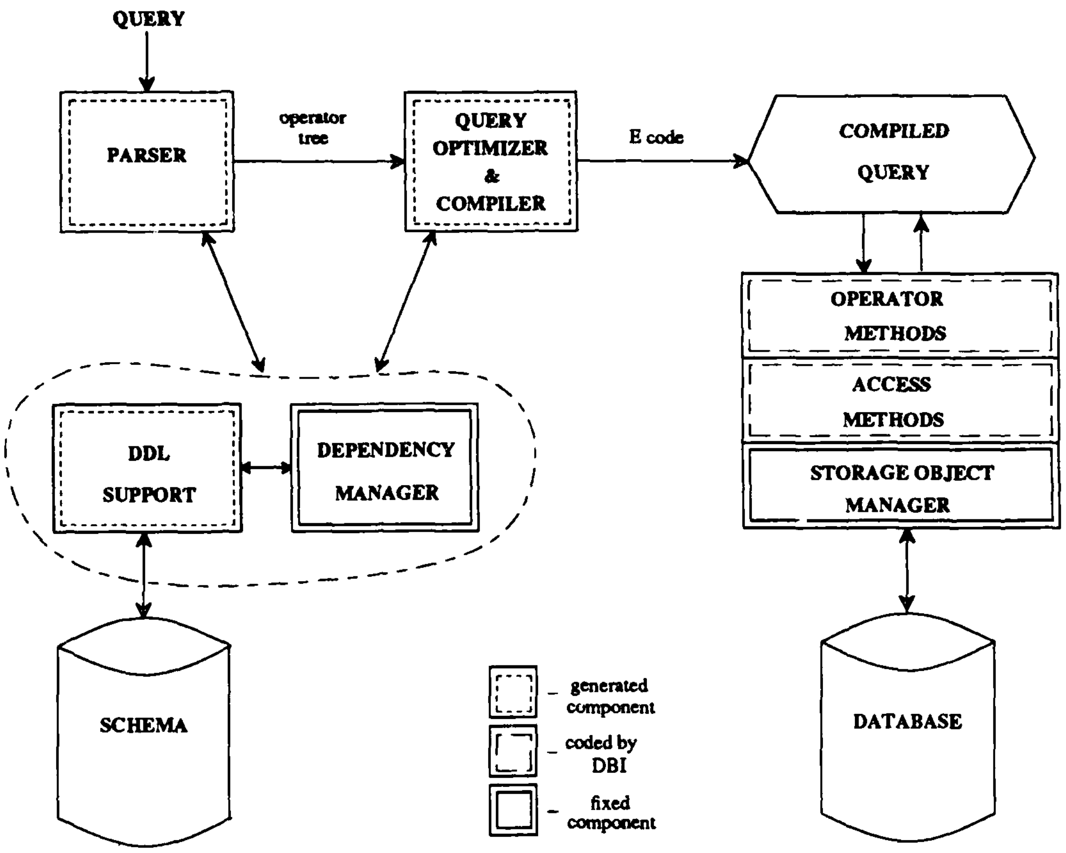

Figure 1: An EXODUS-Based DBMS.

Finally, the top level of the EXODUS architecture consists of a set of components that are generated from DBI specifications. One such component is the query optimizer and compiler. We also plan to investigate and develop tools to automate the process of producing new DML/DDL components, which are the query parser and DDL support components shown in Figure 1. (This idea is similar to the data model compiler notion of [Mary86].) DML components generate operator trees to be fed to the query optimizer, while DDL components produce compiled E code; that is, user-level schema definitions result in the definition of associated E types (which are stored away and registered with the Dependency Manager) and E code to create the associated EXODUS files.

Note that is also possible to use E as a lower-level mechanism for accessing a database directly, for applications needing such low-level access. Assume that one has used the tools provided by EXODUS to construct an application-specific database system. "Normal" accesses to the database would be processed through its ad-hoc or embedded query interfaces, while those applications needing direct access to storage objects would be developed using E. Since schema information for all storage objects is maintained internally in E form, the application programs can access storage objects corresponding to entity instances that were created via the ad-hoc query interface. One could also layer an application program on top of the access methods or operator methods layers without necessarily using the front-end portion of the system. Thus, one shared database can be used by both types of applications with little or no loss of efficiency and minimal loss of data independence. For certain applications, the availability of such a direct interface is critical to obtain reasonable performance [Rube87]. The flexibility of the EXODUS approach to extensible database systems will enable users to customize the system to fit such needs.

3. PROJECT STATUS

At the current time a first version of the EXODUS Storage Manager and E compiler have been completed and integrated with each other. Currently, we are working on a second version of the storage manager that will be able to exploit the capabilities of shared-memory multiprocessors. A number of industrial firms have contacted Wisconsin regarding the acquisition of the storage manager. The EXODUS optimizer generator is a completed piece of software that has been provided to both Bell Labs and Hewlett-Packard Labs for experimentation. An evaluation of the optimizer generator indicates that a rule-based query optimizer produces query plans that are competitive with those produced by commercial query optimizers.

While an operational E compiler now exists, the compiler does not yet optimize the movement of storage objects between disk and main memory. Other important aspects of the compiler that remain to be implemented (which are not, however, viewed as being technically difficult) are support for full interitance, completing the file abstraction provided by the language, and destructors for persistent objects. If E is to be "competitive", the performance of access and operator methods written in E must be comparable with the same algorithms coded in C by "experts" directly against the EXODUS Storage Manager. We are currently studying alternative techniques (such as classical register allocation techniques and loop parallelization strategies) for minimizing loads and stores of objects.

As a demonstration of power of the EXODUS toolkit, during April of this year we produced a demonstration relational database management system using the entire toolkit. The result of this effort was demonstrated at the 1988 SIGMOD conference. Access methods and operator methods were written using E. Relational queries, after being parsed, were optimized using an optimizer generated using the EXODUS optimizer generator and then were translated to E. Finally, queries were compiled and dynamically loaded into the database server for execution. The demonstration was very well received by the attendees of the conference. While this demonstration vehicle served to illustrate the power of the EXODUS toolkit (we put it together in about 4 weeks), it pointed out the need for several additional tools. We will be working on these tools in the coming year as part of putting the EXTRA/EXCESS system together.

We intend to distribute the components of the EXODUS toolkit as they become available. The EXODUS Optimizer Generator has already been distributed to HP Labs and AT&T Bell Labs. We plans to release the second version of the Storage Manager by September 1988 and the first version of the E compiler by December 1988.

Summary of Accomplishments

Accomplishments before FY88:

- Completed overall design of the EXODUS database system.

- Completed design for the EXODUS Storage Manager and began its implementation.

- Completed the design of the E programming language and began its implementation.

- Completed the implementation and performed an evaluation of the EXODUS optimizer generator.

- Completed design and implementation of the EXODUS Dependency Manager.

Accomplishments for FY88:

- Completed a first version of the EXODUS Storage Manager.

- Completed a detailed design of recovery algorithms for the EXODUS storage manager.

- Completed a first version of the E compiler which includes: generators, iterators, and persistence.

- Designed the EXTRA data model and EXCESS query language.

- Produced a demonstration relational database management system using the entire EXODUS toolkit.

Objectives for FY89:

- Begin distribution of the EXODUS Storage Manager

- Complete the design of the EXTRA/EXCESS object-oriented database system and begin its implementa-.ion.

- Investigate query optimization in the context of object-oriented database systems — in particular, within the context of EXTRA and EXCESS.

- Research in performance for complex objects through replication — in particular, within the context of EXTRA and EXCESS.

- Design and evaluate alternative optimization strategies for E and incorporate the best strategy in the E compiler. Also provide full support for inheritance and virtual functions. Implement the "program-fragment" server for E program support.

4. KEY TECHNICAL REPORTS

The Architecture of the EXODUS Extensible DBMS, M. Carey, D. DeWitt, D. Frank, G. Graefe, J. E. Richardson, E. J. Shekita and M. Muralikrishna, Proceedings of the International Workshop on Object Oriented Database Systems, Asilomar, CA., September, 1986

Object and File Management in the EXODUS Extensible Database System, M. Carey, D. DeWitt, J. Richardson, and E. Shekita, Proceedings of the 1986 VLDB Conference, Japan, August 1986.

Programming Constructs for Database System Implementation in EXODUS, J. Richardson and M. Carey, Proceedings of the 1987 SIGMOD Conference, San Francisco, CA, May, 1987.

The EXODUS Optimizer Generator, G. Graefe and D. DeWitt, Proceedings of the 1987 SIGMOD Conference, San Francisco, CA, May 1987.

A Data Model and Query Language for EXODUS, M. Carey, D. DeWitt, and S. Vandenberg) Proceedings of the 1988 SIGMOD Conference, Chicago, Ill., June, 1988.

Persistence in EXODUS, Richardson, J., Carey, M., DeWitt, D., and Shekita, E., *Proceedings of the Appin Workshop on Persistent Object Systems*, Appin, Scotland, August 1987.

Implementing Persistence in E, Richardson, J., and Carey, M., submitted to the *Newcastle Workshop on Persistent Object Systems*, September, 1988.

Persistence in the E Language: Issues and Implementation, Richardson, J., and Carey, M., submitted to *Software Practice and Experience*, September, 1988.

Storage Management for Objects in EXODUS, Carey, M., DeWitt, D., Richardson, J., and Shekita, E., in **Object-Oriented Concepts, Applications, and Databases**, W. Kim and F. Lochovsky, eds., Addison-Wesley Publishing Co., 1988, to appear.

REFERENCES

[Atki78] Atkinson, R., B. Liskov, and R. Scheifler, "Aspects of Implementing CLU," *Proc. of the ACM National Conf.*, 1978.

[Care85] Carey, M. and D. DeWitt, "Extensible Database Systems," *Proc. of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, February 1985.

[Care86a] Carey, M., et al, "Object and File Management in the EXODUS Extensible Database System," *Proc. of the 1986 VLDB Conf.*, Kyoto, Japan, August 1986.

[Care86b] Carey, M., et al, "The Architecture of the EXODUS Extensible DBMS," *Proc. of the Int'l Workshop on Object-Oriented Database Systems*, Asilomar, CA, September 1986

[Cope84] Copeland, G. and D. Maier, "Making Smalltalk a Database System," *Proc. of the 1984 SIGMOD Conf.*, Boston, MA, May 1984.

[Grae87] Graefe, G. and D. DeWitt, "The EXODUS Optimizer Generator," *Proc. of the 1987 SIGMOD Conf.*, San Francisco, CA, May 1987.

[Feld79] Feldman, S., "Make — A Program for Maintaining Computer Programs," *Software — Practice and Experience*, Vol. 9, 1979.

[Lisk77] Liskov, B., et al, "Abstraction Mechanisms in CLU," *Comm. ACM*, 20(8), August 1977.

[Litw80] Litwin, W., "Linear Hashing: A New Tool for File and Table Addressing," *Proc. of the 1980 VLDB Conf.*, Montreal, Canada, October 1980.

[Mary86] Maryanski, F., et al, "The Data Model Compiler: A Tool for Generating Object-Oriented Database Systems," *Proc. of the Int'l Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, September 1986.

[Niev84] Nievergelt, J., H. Hintenberger, and K. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM Trans. on Database Systems*, Vol. 9, No. 1, March 1984.

[Rich87] Richardson, J., and M. Carey, "Programming Constructs for Database System Implementation in EXODUS," *Proc. of the 1987 SIGMOD Conf.*, San Francisco, CA, May 1987.

[Rube87] Rubenstein, W. and R. Cattell, "Benchmarks for Database Response Time," *Proc. of the 1987 SIGMOD Conf.*, San Francisco, CA, May 1987.

[Ston86b] Stonebraker, M., "Object Management in POSTGRES Using Procedures," *Proc. of the Int'l Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, September 1986.

[Stro86] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, Reading, 1986.

Final report
Multicomputer operating systems and applications

Sponsored by the
Defense Advanced Research Projects Agency
Contract N00014-85-K0788

Contractor:

University of Wisconsin—Madison
750 University Avenue
Madison, WI 53706

Principal Investigators:

Raphael A. Finkel
Marvin H. Solomon
1210 W. Dayton St.
Madison, WI 53706
(608) 262-1204

Effective: 21 Aug 1985
Expires: 15 Jan 1988

1. Introduction

This report summarizes the activities supported by DOD contract N00014-85-K0788 and the results of those activities.

Our continuing goal was to characterize the algorithms for which very large multicomputers are appropriate. Multicomputers are one promising direction for the construction of large-scale computing engines. Other directions include multiprocessors (which share memory) and uniprocessors like the Cray that employ pipelining and parallel functional units. The advantages of multicomputers are several.

- It is easy to increase the power of the engine by adding more constituent computers (which we call *machines*).
- Algorithms do not have to worry about memory conflicts between machines.
- No hardware-development research is required to prove the fundamental concepts.

Our research has been carried out largely on the Crystal multicomputer, which was funded by NSF (grant MCS-8105904) to purchase approximately 40 node machines, each a VAX-11/750. These nodes were originally interconnected by a 10 Megabit/sec *ProNet* token ring manufactured by Proteon Corporation. During the period of this contract, the network was upgrades to 80 Megabit/sec. The purpose of this hardware is to promote research in distributed algorithms for a wide variety of applications. In order to provide different applications simultaneous access to the network hardware, Crystal provides a software package called the *nugget* that resides on each node. The nugget enforces allocation of the network among different applications by virtualizing communications within partitions of the network. These partitions are established interactively through a host machine. Interaction between the user and individual machines is provided by the nugget facility of virtual terminals. Initial loading, control, and debugging of programs on node machines is controlled by nugget software.

Crystal is being used for a wide range of applications. Research is underway in distributed operating systems, programming languages for distributed systems, tools for monitoring, measuring, and debugging distributed systems, multiprocessor database machines, parallel algorithms for mathematical programming, numerical analysis and computer vision, and evaluating alternative protocols for high performance local network communications. Some of those projects are discussed in detail here.

Our most recent work has concentrated on experiments involving Charlotte, development of user interfaces, tools for distributed backtracking and parallel debugging, distributed resource allocation, algorithms for the implementation of distributed languages, and an algorithm for parallel solution of a network flow problem.

This broad attack has been broadly successful. Significant advances have been made in most of these areas. In the following sections, we summarize the most recent results and refer the reader to papers in the appendix for details.

2. Charlotte

In the early part of the contract period, the Charlotte operating system stablized and further development of operating system software ceased. The design of Charlotte is presented and discussed in a section of paper describing Crystal, which has appeared in the *IEEE Transactions on Computers* [24]. We have made numerous experiments using Charlotte to implement parallel solutions to a variety of computationally expensive problems. Reports on the results of these experiments are described in three technical reports, which are included as appendices [12, 14, 16]. A more detailed description of interprocess communication facilities in Charlotte has appeared as a technical report [13] and in *IEEE Software* [22]. We have also presented a paper reflecting on lessons learned from the design of Charlotte [15, 26].

3. User Interfaces

The Charlotte command interpreter ("shell") is rather primitive. A project to create a better user interface blossomed into an in-depth research project in the area of user interfaces. The goal of this research shifted from the creation of a particular user interface to tools for creating user interfaces in general. The product of this investigation, called "Dost", assumes the programmer has defined his application as a collection of communicating processes. Each application has a *display manager* that translates between the language of user interaction (editing display objects on the screen) and the language of process interaction (exchanging messages). Thus the user appears to application processes to be another process that can send and receive messages, while the application processes (and the data objects they manage) appear to the user to be objects that can be directly manipulated. The generation of the display manager is automated by instrumenting the compiler to produce tables describing the data types defined by the application. Thus an application can simply display some of its variables and allow the user to edit them, without worrying about translation to and from a textual representations.

Unfortunately, a thorough exploration of these ideas requires a hardware and software environment (a bitmap display, mouse, and window system) that was not available with Charlotte at the time this research was being done. Therefore, Dost was prototyped in the the Xerox Development Environment on Dandelion workstations.

This research is described in detail in a Ph. D. thesis [3], and is summarized in two conference papers [19,28] and a paper that has been submitted for publication in the *ACM Transactions on Programming Languages and Systems* [31].

4. Communications Interface

Our measurements of Charlotte indicated that the principal bottleneck limiting performance was the processing overhead associated with transmitting a message. A study of ways to mitigate this problem culminated in the design of an alternative architecture for individual Charlotte nodes. The study had four phases. The first phase carefully measured a variety of message-based operating systems, including Charlotte. In the second phase, we used these measurements to design a hardware and software architecture tailored to minize the computational latency. The third phase produced a working prototype of the software design on a shared-memory multiprocessor. This prototype demonstrated the feasibility of the design, and allowed us to make detailed measurements of the load it produced on the hardware. The fourth phase consisted of a construction of mathematical models of various hardware architectures, using the formalism of ***generalized timed Petrie nets.*** These models were solved, using a software package produced by other researchers at the University of Wisconsin. The results nicely match the experimental measurements derived in phase three, and indicated that the proposed hardware architecture could achieve significant improvements.

This research is reported in detail in a Ph.D. dissertation [4]. Various aspects of it were reported in conferences [25,27], and have been submitted to journals [32].

5. Distributed Backtracking

One of the principal investigators (Finkel) in cooperation with another faculty member (U. Manber) developed a special-purpose tool for parallelizing applications that can be presented as backtracking applications. This tool, called *DIB* (for Distributed Implementation of Backtracking), was implemented directly on top of the Crystal nugget (that is, not using Charlotte). Results of experiments using this tool are presented in the reports cited earlier. In addition, descriptions of DIB and the more interesting implementation problems have appeared as a technical report [11], a conference paper [18], and a journal paper [23]. DIB was subsequently ported to a shared-memory multiprocessor. It also inspired another tool, called *PIB* (for Parallel Implementation of Backtracking) specifically designed for a multiprocessor. Research involving both of these tools continues.

6. Parallel Debugging

A research project concerning tools for debugging in a distributed environment that was begun before the commencement of this contract was completed and lead to a Ph. D. thesis [2]. Summaries of the major results were presented at a workshop [20], and in a published paper [30].

7. Distributed Allocation

Research in distributed resource allocation, begun under a predecessor of this contract, led to a Ph.D. thesis. A journal article presenting a particularly interesting algorithm that arose from this research has been published [21].

8. Miscellaneous Other Results

An idea that arose in the implementation of the Lynx language (which was developed under the predecessor to this contract) has been published [29]. Dr. Finkel, working in cooperation with other researchers, has developed a parallel algorithm for the solution of a problem in network flows [17].

9. Bibliography

A list of publications created with support from this contract follows. Selected publications, indicated in the following list by an asterisk, are appended.

9.1. Doctoral Dissertations

[1] M. L. Scott, "Disign and implementation of a distributed systems language," Technical Report 596, Computer Sciences Deparment, University of Wisconsin—Madison (May, 1985).

[2] A. J. Gordon, "Ordering errors in distributed programs," Technical Report 611, Computer Sciences Deparment, University of Wisconsin—Madison (August, 1985).

[3] P. Dewan, "Automatic generation of user interfaces," Technical Report 666, Computer Sciences Deparment, University of Wisconsin—Madison (September, 1986).

[4] U. Ramchandran, "Haruware support for interprocess communication," Technical Report 667, Computer Sciences Deparment, University of Wisconsin—Madison (September, 1986).

[5] B. Rosenburg, "At tomatic generation of commication protocols," Technical Report 670, Computer Sciences Deparment, University of Wisconsin—Madison (October, 1986).

[6] Y. Artsy, "A study of fully open computing systems," Technical Report 769, Computer Sciences Deparment, University of Wisconsin—Madison (1987).

[7] C.-Q. Yang, "A structured and automatic approach to the performance measurement of parallel and distributed programs," Technical Report 713, Computer Sciences Deparment, University of Wisconsin—Madison (August, 1987).

[8] H.-Y. Chang, "Dynamic scheduling algorithms for distributed soft real-time systems," Technical Report 728, Computer Sciences Deparment, University of Wisconsin—Madison (November, 1987).

[9] P. E. Krueger, "Distributed scheduling for a changing environment," Technical Report 780, Computer Sciences Deparment, University of Wisconsin—Madison (June, 1988).

9.2. Technical Reports

[10] Y. Artsy, R. A. Finkel, and H.-Y. Chang, "Processes migrate in Charlotte," Technical Report 655, Computer Sciences Deparment, University of Wisconsin—Madison (??).

[11] R. A. Finkel and U. Manber, "DIB—A distributed implementation of backtracking," Technical Report 588, Computer Sciences Deparment, University of Wisconsin—Madison (March, 1985).

[12] *R. A. Finkel, A. Anantharam, S. Dasgupta, T. Goradia, P. Kaikini, C. Ng, M. Subbarao, G. Venkatesh, S. Verma, and K. Vora, "Experience with Crystal, Charlotte, and Lynx," Technical Report 630, Computer Sciences Deparment, University of Wisconsin—Madison (February, 1986).

[13] Y. Artsy, H.-Y. Chang, and R. A. Finkel, "Interprocess communication in Charlotte," Technical Report 632, Computer Sciences Deparment, University of Wisconsin—Madison (February, 1986).

[14] *R. A. Finkel, B. Barzideh, C. W. Bhide, M.-O. Lam, D. Nelson, R. Polisetty, S. Rajaran, I. Steinberg, and G. A. Venkatesh, "Experience with Crystal, Charlotte, and Lynx: Second Report," Technical Report 649, Computer Sciences Deparment, University of Wisconsin—Madison (July, 1986).

[15] R. A. Finkel, M. L. Scott, W. Kalsow, Y. Artsy, H.-Y. Chang, P. Dewan, A. J. Gordon, B. Rosenburg, M. Solomon, and C.-Q. Yang, "Experience with Charlotte: Simplicity versus function in a distributed operating system," Technical Report 653, Computer Sciences Deparment, University of Wisconsin—Madison (July, 1986).

[16] *R. A. Finkel, G. Das, D. Ghoshal, K. Gupta, G. Jayaraman, M. Kacker, J. Kohli, V. Mani, A. Rajhaven, M. Tsang, and S. Vauapeyam, "Experience with Crystal, Charlotte, and Lynx: Third Report," Technical Report 673, Computer Sciences Deparment, University of Wisconsin—Madison (November, 1986).

[17] M. D. Chang, M. Engquist, R. A. Finkel, and R. R. Meyer, "A parallel algorithm for generalized networks," Technical Report 642, Computer Sciences Deparment, University of Wisconsin—Madison (March, 1987).

9.3. Refereed Publications

[18] R. A. Finkel and U. Manber, "DIB—A distributed implementation of backtracking," *Proceedings of the Fifth International Conference on Distributed Computing Systems*, pp. 446-452 (May 1985).

[19] P. Dewan and M. Solomon, "An Approach to Generalized Editing," *Proceedings of the IEEE First International Conference on Computer Workstations*, pp. 52-60 (November 1985).

[20] A. J. Gordon and R. A. Finkel, "TAP: A distributed debugger for handling timing errors," *Proceedings of a Workshop on Software Testing*, (July 1986).

[21] *R. A. Finkel and H. H. Madduri, "An efficient deadlock avoidance algorithm," *Information Processing Letters* 24 pp. 25-30 (January 1987).

[22] *Y. Artsy, H.-Y. Chang, and R. A. Finkel, "Interprocess communication in Charlotte," *IEEE Software* 4(1)(January 1987).

[23] *R. A. Finkel and U. Manber, "DIB—A distributed implementation of backtracking," *ACM Transactions on Programming Languages and Systems* 9(2) pp. 235-256 (April 1987).

[24] *D. J. DeWitt, R. A. Finkel, and M. Solomon, "The Crystal multicomputer: Design and implementation experience," *IEEE Transactions on Software Engineering*, pp. 953-966 (August 1987).

[25] U. Ramachandran, M. Solomon, and M. Vernon, "Hardware support for interprocess communication," *14th International Symposium on Computer Architecture*, (June 2-5, 1987).

[26] Y. Artsy and R. A. Finkel, "Simplicity, efficiency, and functionality in designing a process migration facility," *Second IEEE Israel Conference on Computer Systems and Software Engineering*, (May 1987).

[27] U. Ramachandran, M. Solomon, and M. Vernon, "Techniques for reducing model complexity of large systems," *International Conference on Parallel Processing*, (August 17-21, 1987).

[28] P. Dewan and M. Solomon, "Dost: An Environment to Support Automatic Generation of User Interfaces," *Proceedings of the Second ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, (December 1986). In *ACM SIGPLAN Notices* 22:1 (January 1987)

[29] M. L. Scott and R. A. Finkel, "A simple mechanism .or type security across compilation units," *IEEE transactions of Software Engineering*, (August 1988).

[30] A. J. Gordon and R. A. Finkel, "Handling timing errors in distributed programs," *IEEE Transactions on Software Engineering* 14(10)(October 1988).

[31] *P. Dewan and M. Solomon, "An Approach to Support Automatic Generation of User Interfaces," *ACM Transactions on Programming Languages and Systems*, (1988). Submitted for publication.

[32] U. Ramachandran, M. Solomon, and M. Vernon, "Hardware support for interprocess communication," *IEEE Transactions on Computers*, (1988). Submitted for publication.